Fast Terrain Rendering Using Geometrical MipMapping

Willem H. de Boer, *whdeboer@iname.com*
E-mersion Project, *October 2000*, http://www.connectii.net/emersion

1 INTRODUCTION

There is a variety of algorithms out there that speed up rendering of terrain-data, varying from simple quadtree based frustum-culling of unseen terrain chunks to using occluder sets to using (continuous) Level of Detail. All of these techniques serve the purpose of reducing the amount of triangles to be pushed through the rendering pipeline. Many terrain rendering algorithms use a combination of several of these techniques to even further speed up rendering. Famous examples are the ROAM algorithm [3], Lindstrom *et al.* SIGGRAPH '96 paper on terrain rendering [1], and NDL's approach to terrain rendering to be used in real-time games [2]. Most of these algorithms were invented (long) before hardware rendering became the industry's standard, and therefor may not be suitable to be used in conjunction with 3d hardware rendering anymore. Therefor, new algorithms must be found that will give the best results when used together with 3d hardware rendering. Because 3d hardware is able to process and render a large amount of triangles per frame, the algorithm can resort to more *conservative* culling methods, thereby not necessarily delivering the 'perfect set' of render-data, but pushing *as much triangles* through the pipeline as hardware can handle, with *the least amount of CPU overhead*. I have come up with a method which complies to the above said and - as far as I know - has not been used before. The algorithm has been implemented as part of the E-mersion project. This paper will give a full description of the algorithm.

2 OVERVIEW

This section describes the complete algorithm cut up into three distinct parts. The first part will give an outline of the representation of terrain-data, how it is organized in memory and how it will eventually be rendered. Section 2 talks about basic frustum culling of terrain 'chunks', and the last section describes GeoMipMaps using the standard mipmap technique for textures as an analogy.

2. 1 Terrain-data representation

Representing the terrain can be done in several ways, but I have chosen to use the one that is used by the majority of today's terrain-rendering engines. The reason I have chosen this representation is that it 1) is simple to implement, 2) requires minimal hard-disk storage, 3) is suitable for GeoMipMaps, as is explained in section 2.3. One disadvantage of this representation (which is basically a 2d solution) is that it does not support 'overhangs'.

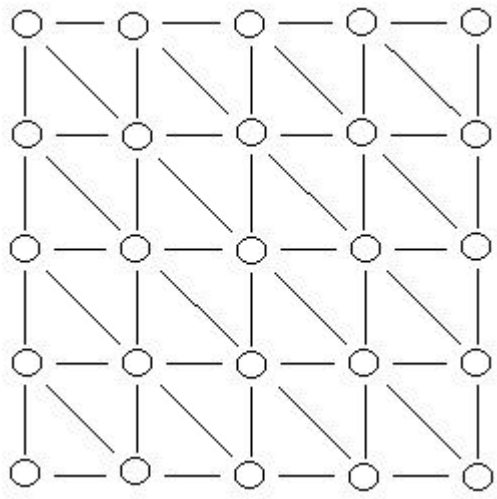

Figure 1: Top-view of a terrain block mesh representation. The white circles represent the mesh's vertices. The lines represent the connections between vertices. Note that each quadrilateral (or quad) is built up out of 2 triangles.

The terrain is laid out in a grid of vertices with a fixed distance between eachother. The horizontal and vertical number of vertices in the grid must be of the form 2^n+1, where $n \in [1,\rightarrow)$. This will result in 2^n quadrilaterals, with 4 neighbouring vertices acting as its corners. Each quadrilateral consists of 2 triangles, which are used as the draw-primitives that are eventually sent to the graphics pipeline.

Each vertex' x and z components are set to a fixed value, which will not be altered during the entire process. The vertex' y component represents the terrain's height-value at that particular position, which is read - at terrain-load time - from a 8-bit gray-value bitmap file that has the exact same dimensions as the terrain grid. The entire terrain grid is then cut up into, what I call, terrain blocks which have a fixed size and must be of the form 2^n+1 ($n \in [1,\rightarrow)$). This is a preprocessing step for the quadtree frustum-culling stage which will be explained later, and these blocks will also serve as the level 0 primitives for GeoMipMaps (see section 2.3). Figure 1 shows such a terrain block, which is 5x5 vertices. The actual size of the terrain block can be chosen freely. In the E-mersion implementation this is 17x17, which I have found to be the fastest considering my total terrain-grid size of 257x257. Experimentation is of great necessity here.

One advantage of having this particular terrain block layout is that one such block can be optimized for rendering, using one draw-primitive call for the entire block and, even better, using indexing to get rid of multiple transformation of vertices. A small disadvantage of terrain blocks is that the vertices of each of the 4 edges are shared by the neighbouring terrain blocks and therefor will be transformed twice.

2.2 Basic view-frustum culling

Because large parts of the terrain will not be visible (i.e. not inside the view-frustum) from a certain camera point, they need not to be rendered and therefor should be culled away early to prevent unnecessary calculations. A method which has proven to be quite effective for fast culling is a data structure called a quadtree. I am not going to explain quadtrees here, because there exist many good tutorials/papers/articles on them on the internet.

The quadtree is generated at terrain load time, and consists of 3d boundingboxes only. The size of the terrain together with the size of the terrain blocks decide the eventual depth of the quadtree. Quadtree nodes consist of 3d boundingboxes which physically contain the node's entire sub-tree's boundingboxes, where - at the leaf - the actual terrain block's bounding box can be found. Each leaf has a reference to the terrain block that it is boundingbox contains. The reason quadtrees are used over octrees is that the terrain layout is essentially 2d, and therefor we can suffice with a 2d spatial organization scheme.

Terrain blocks are marked visible when the leaf's boundingbox is at least partially inside the view-frustum. After having completed descending the tree, starting culling at the tree's rootnode and proceeding from there on, marking terrain blocks either visible or invisible, we are left with a set of terrain blocks which can be immediately sent to the graphics pipeline. Because terrain complexity can be quite high, we will not get the desirable framerate unless the terrain is very small; the above described method will not suffice for high complexity (eg, a large grid) terrain. This is where I introduce a new term: GeoMipMaps. They make this terrain rendering method unique and suitable for 3d hardware rendering as I will describe next.

2.3 The Texture Mipmap Analogy (introducing GeoMipMaps)

Terrain blocks that are far away from the camera do not need to be rendered with the same detail as terrain blocks that are near the camera. They can be approximated by a lower resolution version, thereby drastically decreasing triangle count and increasing render speed. This is called Level of Detail, a term which is famous among many, and is used in many of today's terrain rendering algorithms. Many of today's algorithms for fast rendering of terrain data use some sort of Level of Detail scheme, although many reside to per-triangle methods which are not suitable for use in conjunction with 3d hardware rendering. To reduce CPU performance overhead, we should perform Level of Detail on a higher level, and this is where GeoMipMaps kick in.

Consider the ordinary mipmapping technique for textures [5]. A chain of mipmaps is created for each texture, the first item in the chain is the actual texture with each succeeding item being the previous item scaled down to half its resolution, until a desired number of items (levels) is reached. When a texture is at certain distance, *d*, from the camera, the appropriate level in the mipmap chain is chosen and used for rendering instead of the actual high resolution texture. We can apply this concept to three-dimensional meshes too, where the high resolution texture's 3d equivalent is the terrain block, and mipmaps are calculated by scaling down the terrain block. Figure 2 shows a terrain block and its direct successor in the mipmap chain. This chain can, ofcourse, be extended by another item in the chain, which is of even lower resolution. We can precalculate and store these GeoMipMaps in memory preferably at terrain-load time.

From now on, this paper will use the term GeoMipMap level 0 for the default resolution terrain block, and GeoMipMap level N ($N \in [1,\rightarrow)$) for each lower resolution version.

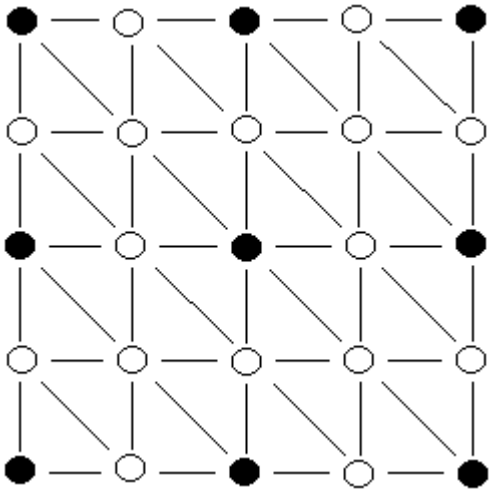

Figure 2: Default resolution of terrain block mesh on the left (GeoMipMap level 0), and a lower resolution version (GeoMipMap level 1) on the right. Note that a GeoMipMap level 2 can also be generated.

2.3.1 Choosing the appropriate GeoMipMap level

Deciding which GeoMipMap level to use for which distance (***d***) is something I would like to explain in this section. Using a preset (fixed) ***d*** for each GeoMipMap level will result in unwanted popping artifacts. Switching from one level to another at a fixed ***d*** without any condition will result in a sudden change in geometry; vertices being added at a decrease in GeoMipMap level, and vertices being omitted at an increase in GeoMipMap level. Have another look at the Figure 2. When switching from level 0 to level 1 the white vertices of level 0 will be removed to get level 1. This gives a wrongness to the overall terrain, and looks incorrect when viewed up close. When looking at texture mipmapping, the algorithm only chooses a lower resolution mipmap when the current mipmap's pixel-to-texel ratio is no longer 1:1, this occurs at a certain ***d*** [5]. The exact computation for choosing the appropriate level resides in hardware, and really does not matter to us now. We can apply the same kind of concept to GeoMipMaps to reduce popping artifacts.

When switching from GeoMipMap level 0 to 1, the decrease in detail will give a wrongness or error to the terrain block. This is caused by the the removal of vertices, which will give a change in height δ as shown in figure 3. This δ will be less noticable the greater ***d*** will become, because of perspective. We can also project δ to its equivalent length in screen space pixels (which we

will call ε), which is eventually what the user will notice. When ε exceeds a certain threshold τ of, say, 4 pixels, the error will be too noticable and therefor it is not permitted to switch to a higher GeoMipMap level until ε is smaller than τ. But because every GeoMipMap level consists of several of these errors (one for each vertex that has been removed), we must somehow calculate which δ to take to project to screen-space pixels. By taking the resulting δ from $max\{\delta_0,\ldots, \delta_{n-1}\}$ (where *n* is the number of δ's in the GeoMipMap) to calculate ε for, we have covered the worst-case scenario. If this particular value is lower than τ, then *all* the GeoMipMap's error value's will be smaller than τ. And vice versa, if this value is higher than τ, there will be *at least* one error that is too noticable (eq, is higher than τ). We choose this δ at GeoMipMap creation, and store it. From now on, if δ or ε is being mentioned, the GeoMipMap's $max\{\delta_0,\ldots, \delta_{n-1}\}$ or $max\{\varepsilon_0,\ldots, \varepsilon_{n-1}\}$ is is meant (*n* is number of δ or ε values in GeoMipMap).

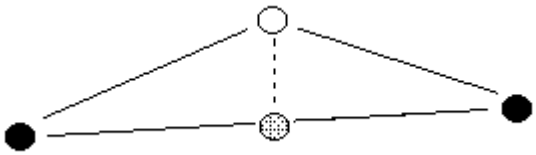

Figure 3: Vertex error as seen from the side. The dotted line represents the geometric height-change (δ) which will occur when removing the white vertex.The exact height-change value can be calculated by substracting the white vertex from the gray vertex. After removing the white vertex, the vertex' position, will be at the imagenary gray vertex.

In order to find out whether it is appropriate to switch to a higher GeoMipMap level for a given ***d***, we must compare the current GeoMipMap's ε to τ. If inequality ε > τ is true, then switching to a higher level will result in too great of an error. If it is does not, then it is allowed to move one level up. Note that the next GeoMipMap's level ε *is also lower* than τ, in which case we can use *that* GeoMipMap level, and so on, until the right level is reached.

The correct inequality to solve the above mentioned condition can be found in [1]. A 3d-space equivalent, that uses a 3d-equivalent of τ, can be found in [4]. The formula mentioned in [1] calculates the exact value for ε from any given point of view relative to the GeoMipMap. The higher the slope of the camera's view vector *relative* to the GeoMipMap, the smaller ε will be, and the higher the probability of using the higher level GeoMipMap will be. This means that, if the camera's direction vector is parallel to the x/z plane (i.e., is horizontal) ε will be at its highest. We can use this fact to speed up our GeoMipMap level decision algorithm.

2.3.1.1 Speeding up GeoMipMap level decision

Because of the ability of 3d hardware rasterizers to render a large amount of triangles, we do not need to calculate the 'perfect-set' of triangles anymore. If using a less exact algorithm will result in less CPU overhead, it is best to use it, as the extra amount of triangles that are generated does not stand up to the fact that it required less CPU time. The goal is to send as much triangles to the hardware as it can handle.

Back to GeoMipMap level decision. Calculating ε and comparing it to τ for *every visible GeoMipMap every frame* can be very processor intensive. Because, according to [1] and [4], ε depends on ***d*** *and* the position of the camera *relative* to the position of the GeoMipMap, there is no way to precalculate ε for each ***d*** and putting it into a Look-Up Table (LUT). But if we treat the camera's direction vector as being permanently horizontal (i.e. not vertical slope), we *can* precalculate ε. This will not deliver precise results for cases where the vertical slope of the camera's view vector relative to the GeoMipMap is high; it will generally use too high detail for these high slope cases, but when reading the previous paragraph it should not be a problem. We have essentially brought down CPU overhead to a minimum.

To even further speed up this process, at terrain-load time an extra data field in each GeoMipMap level is computed, which is set to the *minimum* ***d*** (***D***) at which this level can be used. Now when choosing the appropriate level for each visible GeoMipMap each frame, it suffices to compare ***d*** to $\boldsymbol{D_n}$ (where *n* is the GeoMipMap's level). See Figure 4 for pseudo-code.

```
For Each GeoMipMap Level N
    Compare L to D_N
    if (L > D_N) store to RESULT
End For
return RESULT
```

Figure 4: Pseudo-code for choosing the appropriate GeoMipMap level, where L is the distance from the camera to the center of the GeoMipMap in 3d, and D is $\boldsymbol{D_n}$.

2.3.1.2 Pre-calculating *d*

To precalculate the $\boldsymbol{D_n}$ for each GeoMipMap level, use the equation shown in Equation 1. This equation takes δ as a parameter and returns the appropriate $\boldsymbol{D_n}$. Note that it is advisible to have a symmetrical view-frustum, which should be set up using the (*l, r, b, t, n, f*) view-frustum parameters.

$$\boldsymbol{D_n} = |\delta| \cdot \boldsymbol{C}$$

Equation 1: Use this equation to calculate D_n *for δ. C is a constant which can be calculated using the equation found in Equation 1a)*

$$\boldsymbol{C} = \frac{\boldsymbol{A}}{\boldsymbol{T}}$$

Equation 1a), used to calculate C. A and T are constants which can be calculated using the equations found in 1b) and 1c) respectively.

$$\boldsymbol{A} = \frac{\boldsymbol{n}}{|\boldsymbol{t}|}$$

Equation 1b), used to calculate A. n is the near clipping-plane, and t is the top coordinate of the near clipping-plane, as in (l, r, b, t, n, f).

$$T = \frac{2 \cdot \tau}{v_{res}}$$

Equation 1c), used to calculate T. v_{res} is the vertical screen resolution in pixels.

To save a square-root instruction every time the distance from the current camera to a GeoMipMap must be calculated in real-time, Equation 1 can be rewritten as shown in Equation 2. Pre-calculating D_n^2 will reduce the real-time distance GeoMipMap to camera calculation to $d^2 = (e_x - c_x)^2 + (e_y - c_y)^2 + (e_z - c_z)^2$, where e is the vector $[e_x, e_y, e_z]$ which is the current position of the camera, and c is the vector $[c_x , c_y , c_z]$ which is the center of the current GeoMipMap being processed.

$$D_n^2 = \delta^2 \cdot C^2$$

Equation 2: Equation 1 squared, saving a square-root instruction for every d calculation per GeoMipMap per frame.

2.3.2 Solving geometry-gaps

When two neighbouring GeoMipMaps in the terrain have different levels of detail, cracks at the edges of the GeoMipMaps will occur. This is not desirable for any application of terrain rendering, and should be avoided.
Edges of GeoMipMaps with more detail (i.e., more vertices) contain extra heightmap information as opposed to edges of GeoMipMaps with less detail, and when these two GeoMipMaps share edges, terrain geometry-gaps will show up. Solving this problem essentially implies a re-arrangement of vertex connections so that edges tightly fit with eachother.

One way to solve this, is to add the extra vertices at the edge of the lower detail level of the two neighbouring GeoMipMaps so that it will fit with the highest detail GeoMipMap's edge. This implies that the default vertex layout and connectivity of vertices for the low detail GeoMipMap level must be changed dynamically, and must be updated whenever the neighbouring GeoMipMap changes levels. An extra copy of the original GeoMipMap must be made and stored in memory. This is a slow and memory consuming task, and so this method should be avoided.

Another way to solve this, is to level those vertices of an edge of the higher detail GeoMipMap that add to the detail, so that it will fit the lower detail GeoMipMap's edge. This will cause so called T-junctions, or T-vertices. Because of floating-point inaccuracies, these will deliver a very noticable 'missing pixels' effect.

I have found a good way of solving terrain geometry-gaps or cracks, which does not alter the GeoMipMap level's vertex layout, and will not produce the 'missing pixels' effect. The *only* thing it does is changing connectivity (or indexing) of vertices for the higher detail GeoMipMap. Have a look at Figure 5. You will see the higher detail GeoMipMap which shares its left edge with a (not showed in figure 5) lower detail GeoMipMap. The gray vertices carry the additional heightmap data, which causes the cracks. By omitting these vertices from being 'connected', the edge will seamlessly fit with the lower detail GeoMipMap. A fast way of rendering this is by using two triangle-fans for this edge, emerging from the top-left black vertex, and the black vertex directly under it. The remaining vertices are drawn in the normal way. This process must be performed for each of the four edges which connect the GeoMipMap to a lower detail GeoMipMap. When implementing the algorithm, this means that every GeoMipMap level 0 must have references to all of its four neighbours.

A downside of this method is that, when changing connectivity of vertices, a slight change in (gouraud) shading will occur. This sometimes can be spotted in real-time, but it is only slightly noticable.

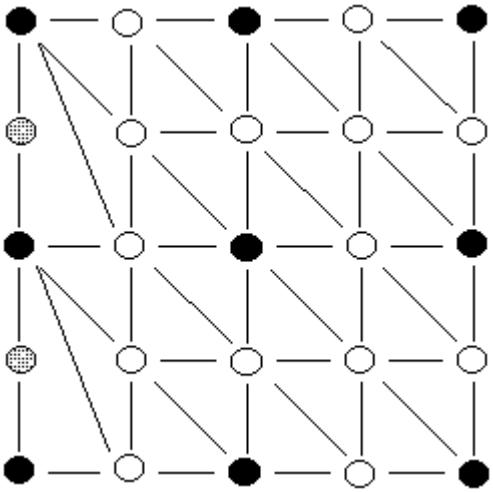

Figure 5: Solving T-junctions and terrain geometry-gaps between a higher level (lower resolution) left-neighbour GeoMipMap. Note that, although the vertices marked gray are still part of this GeoMipMap's mesh, they are not being rendered because of the use of a triangle fan. Omitting rather than adding vertices, has the advantage of not having to change anything for any GeoMipMap. They can remain the same, the only thing that changes is the connection data.

3 EXTENSIONS TO THE BASIC APPROACH

Although the basic approach delivers results which will be satisfactory for use in applications such as real-time fast-action games, there may be some applications that need a more sophisticated solution to two problems, which I will describe and give a solution to in the next two sections.

3.1 Trilinear GeoMipMapping

Although using screen-space pixel errors to select GeoMipMap levels will deliver a tremendous decrease in popping artifacts, there will still be some occasional popping, especially when τ is set to a high value. Although setting τ to 1 or less will eliminate popping altogether, it also implies that the probability of using a higher level GeoMipMap (lower detail) will be very low. A certain value of τ has to be found which gives a good balance between keeping low triangle-counts and keeping popping-artifacts to a sufficiently low level. This value is unique for every application. To keep triangle-count low, without *any* popping I will once more use the texture mipmapping analogy, looking at a technique called Trilinear Filtering.

When looking at ordinary texture mipmapping, one can clearly see the borders of where two mipmaps of different levels touch eachother. These borders, or lines if you will, occur at fixed distances from the camera and can be eliminated by using Trilinear filtering. I am not going to explain the details of Trilinear filtering here, but the concept is simple. Instead of choosing a discrete mipmap for a certain $\boldsymbol{d}$ and using this until a $\boldsymbol{d}$ is reached for which a higher level mipmap can be used**,** an interpolation of the mipmap with the mipmap one level higher is calculated using the fractional distance of both mipmap's precalculated $\boldsymbol{d}$. This will give a smooth blend of mipmaps for any value of $\boldsymbol{d}$. This same principle can be applied to GeoMipMaps. Have another look at Figure 2. GeoMipMap level 0 has a certain precalculated $\boldsymbol{d}$ ($\boldsymbol{D_0}$) which indicates at what $\boldsymbol{d}$ this level can replace the preceeding level for rendering. The succeeding level has an equivalent $\boldsymbol{d}$ ($\boldsymbol{D_1}$). If we calculate the fraction between these two, as shown in Equation 3, we can use this fraction as a multiplier (see Equation 4) for the white vertices of GeoMipMap level 0. This will slowly 'morph' level 0's geometry to level 1's until $\boldsymbol{d}$ has reached $\boldsymbol{D_1}$, after which level 0 can be replaced by level 1. This essentially replaces the sudden change (popping) of GeoMipMap levels with a smooth transition between both.

$$\boldsymbol{t} = \frac{(\boldsymbol{d} - \boldsymbol{D_n})}{(\boldsymbol{D_{n+1}} - \boldsymbol{D_n})}$$

Equation 3: Calculating the interpolation fraction t. d is the distance from the camera to the GeoMipMap, D_n is the precalculated d for the current GeoMipMap, and D_{n+1} is the precalculated d for the succeeding GeoMipMap.

$$\boldsymbol{v'} = \boldsymbol{v} - \boldsymbol{t} \cdot \delta_v$$

Equation 4: Used to calculate the new vertex position, v', for white vertex v. δ_v is the height change (geometric error) of v. See figure 3.

Any edge's white vertices that are neighbours with lower detail GeoMipMaps should *not* be multiplied by $\boldsymbol{t}$, as these do not contribute to the GeoMipMap anymore (see section 2.3.2).

3.2 Basic Progressive GeoMipMap streaming

There are certain applications that require visualization of terrain-data which will not fit into memory in its entirety. To solve this problem, only those terrain blocks that need to be rendered or are at least near the camera need to be loaded from disk and stored in memory. Whenever terrain blocks, which are not already stored in memory, become visible, they need to be loaded from disk. This can be done through streaming.

At application startup time, the quadtree is generated for the *whole* terrain. The node's boundingboxes x, and z coordinates can be calculated beforehand, and the y-coordinates need to be read from the terrain-data bitmap file.
We need to step through every terrain block and do the following:

1) Load terrain block
2) Calculate terrain block's boundingbox and store it in appropriate leaf
3) Calculate all GeoMipMap levels for this terrain block
4) Calculate $\boldsymbol{D_n}$ for every GeoMipMap Level and store it in the terrain block's quadtree leaf.
5) Erase all GeoMipMapLevels and proceed step 1) to 5) for the next terrain blocks

After this, we are left with the quadtree of the whole landscape, with every leaf containing all its terrain blocks $\boldsymbol{D_n}$ values. Note that we have not stored any GeoMipMaps.

At run-time, the quadtree's visible leafs' $\boldsymbol{D_n}$ values are compared to the current $\boldsymbol{d}$ and the appropriate GeoMipMap level is created from disk. Whenever $\boldsymbol{d}$ is smaller than the currently loaded GeoMipMap's $\boldsymbol{D_n}$, the appropriate GeoMipMap level is created progressively. This is a brief description of the technique, and can be further extended.

4 REAL-TIME RESULTS

The method described in this paper has been implemented as part of the E-mersion Project. It currently only uses the basic algorithm; features such as explained in section 3 have not been implemented although it should not be quite difficult to integrate this into the E-mersion terrain renderer. Figure 6 shows a screenshot of GeoMipMaps in action. As you can see, terrain blocks that are further away from the viewer are drawn at a higher level in the GeoMipMap chain (lower resolution) *unless* the geometric error is too high even for that distance.

Entire terrain blocks are drawn using one draw-primitive call using indexed vertices to further speed up rendering. In this demo, τ=8.0 is used. An average frame-rate of 50 s^{-1} is reached with about 11,000 triangles per frame, on a PentiumII 434 Mhz with a Diamond Viper 550 and 64 MB RAM.

A runnable demo can be found on my website (URL at the top of this paper) on the main page. 3d rendering hardware is, obviously, required.

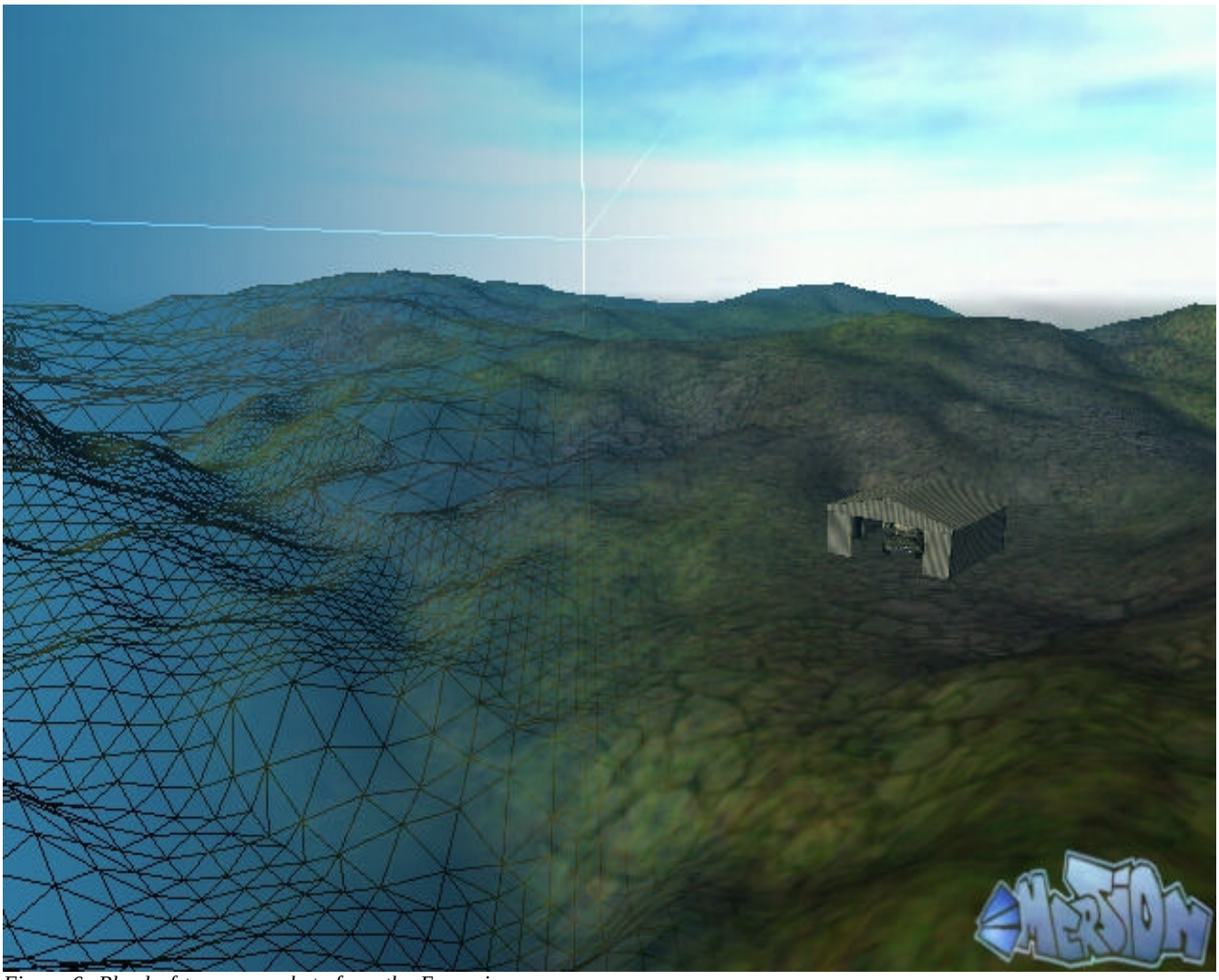

Figure 6: Blend of two screenshots from the E-mersion terrain renderer taken from the same camera point. One screenshot clearly shows GeoMipMaps in action.

5 CONCLUSION

In this paper I have described a method for rendering terrain-data at high speed using 3d rendering hardware. Because of the ability of 3d rendering hardware to render a large amount of triangles, one has to design an algorithm which makes use of this, and is able to process a large amount of triangles while keeping CPU overhead at a low level. This means that the algorithm has to resort to more conservative culling methods thereby not delivering the 'perfect data-set'. The method described in this paper is, in my opinion, suitable for this problem. Preventing per-triangle tests as much as possible, I have basically written an extension to block-based Level of Detail. Block based Level of Detail has proven itself to be very useful and fast for use with 3d hardware rendering, because of its low CPU performance overhead. The noticable popping artifacts - which is the nature of block-based LoD methods - has been reduced by using screen-space pixel errors, and can be totally eliminated using the trilinear 'morphing' method I described. While block based LoD has been used before, I have never seen it being used and compared to texture mipmaps in the way GeoMipMaps do. This method is not too difficult to comprehend and, more importantly, will not result in complex (read: slow) code, which is one of the key factors when designing algorithms to be used together with 3d rendering hardware. The quote *"Keep It Simple, Stupid"* definitely applies to algorithms which make use of 3d rendering hardware.

About the Author

I am currently employed as a Software Engineer at the R&D department of a game-development company based in The Netherlands. If you have any questions regarding this paper or anything about terrain rendering in general, you can contact me. Please send me an e-mail to the address found at the top of this paper. Questions that have already been answered in this paper will not be answered again.

Acknowledgements

Thanks go to Kent Kuné for helping me out with the graphical design aspects of the E-mersion project; to Edward Kmett for giving me helpful advice when solving terrain geometry-gaps; to Simon O'Connor, and Jacco Bikker for taking the time to read and comment on the pre-release version of this paper; and to Lourens Veen for making it possible for this document to be be viewed as a Portable Document Format file.

References

[1] *Real-Time, Continuous Level of Detail Rendering of Height Fields*. Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hodges, Nick Faust, and Gregory A. Turner. In Proceedings of ACM SIGGRAPH 96, August 1996, pp. 109-118.
http://www.cc.gatech.edu/gvu/people/peter.lindstrom/

[2] *The NetImmerse Terrain System***.** Dave Eberly, Director of Engineering, Numerical Design, Ltd.
http://www.ndl.com/terrainwhitepaper.html

[3] *ROAMing Terrain: Real-time Optimally Adapting Meshes.* Mark Duchaineau, LLNL, Murray Wolinksy, LANL, David E. Sigeti, LANL, Mark C. Miller, LLNL, Charles Aldrich, LANL, Mark B. Mineev-Weinstein, LANL.
http://www.llnl.gov/graphics/ROAM/

[4] *Continuous LOD Terrain Meshing Using Adaptive Quadtrees.* Thatcher Ulrich, Slingshot Game Technology.
http://www.gamasutra.com/features/20000228/ulrich_01.htm

[5] *Real-Time Rendering***.** Tomas Möller, Eric Haines. A K Peters, Ltd. ISBN 1-56881-101-2